J. LOGIC PROGRAMMING 1994:19, 20:1–679

W2C: A WAM TO C ASSEMBLER

D. E. STEVENSON

▷ We describe an assembler that accepts `WAM` commands as described in Aït-Kaci. The output of is a program in `C`. We describe how we make the backtracking model of the `WAM` work within the `C` paradigm. We also describe commands that are used to convert `WAM` tagged memory values to be compatible with more "vanilla" `C`. ◁

1. Introduction

Computational science and engineering is the conduct of science and engineering using computational means. Computational science and engineering is an emerging discipline, one that is inherently multidisciplinary. By looking at its roots, one can see that computational programs are likely to be `Fortran` or `C`. But if any discipline begs for a sound logical basis it is computational science and engineering.

What would this foundation look like? In the first place, one must make a decision whether to emulate current practice or strike out on ones own. Current practice, in this author's opinion, stands too far from the constructive roots of numerical analysis and pre-formalist mathematics. Even within that, there is much latitude. The author has chosen the "path less traveled" by attempting to work within the constructive framework of Errett Bishop[2].

To make such a program work, one must have a framework to build on. After some attempts at various ideas, the author has settled on a logic programming paradigm. For the most part, `prolog`ish ideas seem sufficient for now. However, some areas needed expansion. For example, it is necessary to have multiprecision integers and floats. To make a long story short, it was decided to consider the `WAM` as the foundation, even though there were many competing concepts in this area. The simple road would have been to implement an interpreter or use an off-the-shelf system.

The main drawback here is the audience: computational scientists are notoriously conservative (with good cause) and very performance oriented. One also has to see this as a long range conversion mission, which then dictates that the system must

Address correspondence to Department of Computer Science, Clemson University, Clemson, SC 29634-1906. email: steve@cs.clemson.edu

655 Avenue of the Americas, New York, NY 10010 0743-1066/94/$7.00

Register	Use
`A`	Argument register(s) on the stack.†
`B`	Previous Choice Point.
`BP`	Next clause for retry. (not used)
`B0`	`cut` pointer.
`CE`	Continuation environment.
`CP`	Continuation code.
`CPA`	Pointer to current `A` vector.‡
`CF`	Calling function.‡
`E`	Environment stack top.
`H`	Heap top.
`HB`	Highest address requiring `trailing`.
`P`	Program counter.
`R`	Return value register.‡
`S`	Cursor for unification stack.
`TR`	Trail pointer.
`W`	Unprotected argument bank.‡
`X`	Temporary variable bank.†
`Y`	Permanent variable bank.
`Z`	Internal calls to `i`-interpreter.‡

† These are changes to usual description. ‡ New registers.

TABLE 2.1. Named Registers.

be able to use programs developed in other languages. The final decision, then, is to build a compiler that can deal with logical information about numerical and mathematical elements but one that would generate `C` code so that the interface and performance problems would disappear.

We describe the first step—the development of an assembler that accepts `WAM` statements, with some augmentation, and generates standard ANSI `C`. In Section 2 we describe how memory organization. Section 3 describes the development of the individual `WAM` commands, including the `dispatcher` used to control `calls`, *etc.* Finally, we describe the assembler pseudo instructions in Section 4

2. The Memory

(High Address)
Push Down Stack for Unification
Environment/Choice Point Stack
Heap
Atomic Elements (Low Address)

FIGURE 2.1. Standard Memory Layout.

As with any implementation, the layout of memory is crucial. Most of the real problems were solved in the 1980s[3]. The problems of structure sharing and unification without garbage collection leads one to have a word-oriented memory structure called *registers* as show in Figure 2.1.

Our implementation uses a tagged memory system as one would expect. For debugging purposes, we do not limit ourselves to using the two bits that are available because of word oriented (mod 4) addressing. However, we realize that we drastically change the addressing range of the machine by using high-order tags. This couples with another problem discussed in when describing *atoms* (See Section 2.2). Three bits are currently reserved for tags. For portability sake, all tag/value operations are `C` macros. This constitutes the "microcode" and this microcode is available through the compiler.

Even with the understanding produced in [3], the `WAM` described in [1] has some "magic" banks of registers. In particular the `A`, `X`, and `Y` banks took quite a few attempts to get correct. In our implementation, these three banks are special areas of the environment stack. The various memory areas are accessible through "microcode" using the naming convention of `get`*Areaname* and `put`*Areaname*. The special coercions are also "microcode." The `PDL` operations are subroutines, as would be expected. *Implementation Note.* The system was developed using an interpreter for the memory operations. This was termed the `i`-machine. In the long run, this saved much time and effort as errors in understanding could be caught through debuggers. `gdb` and other `C` debuggers are really inadequate when working in highly macro-ized systems.

It turned out to be helpful to add several special registers and one special bank. The `CPA` is required to define state properly and `R` register is a global result register used by the `i`-machine. `R` holds over as an artifact in the macro-ized implementation.

2.1. Definition of Variables

Variables are straightforward since everything in the system is word oriented. The unbound variable is a pointer to itself. This means, however, that everything must be passed *by reference*. This in turn means that the procedure `deref` must be used everywhere. This leaves the optimization question of making `deref` into an *inline* procedure. Actually assignments to variables are accomplished by the various `WAM` instructions as outlined in [1].

2.2. Atoms

There has been some hesitancy on how to handle atoms, particularly given the functional bent of numerical evaluation. Using multiprecision integers, rationals, and floats considerably complicates storage considerations. The undefined length nature of character strings is also a complication. None of these concepts is addressed in [1]. In fact, a deficiency of the model is its handling of non primitive types. Our approach is straightforward: the lowest portion of memory is set aside as the `atom` area. These are `registers` that contain a pointer to the data. Therefore, the data itself may reside anywhere so long as that address need not be used directly. This representation was chosen so as to not impose more tags, further restricting the addressing range.

Get	Put	Set	Unify
get_variable	put_variable	set_variable	unify_variable
get_value	put_value	set_value	unify_value
	put_unsafe_value		
get_structure	put_structure		
get_list	put_list		
get_constant	put_constant	set_constant	unify_constant
		set_local_value	unify_local_value
		set_void	unify_void

Choice	Cut	Ancillary	Control
try_me_else	neck_cut	backtrack	allocate
retry_me_else	get_level	deref	deallocate
trust_me	cut	bind	call
try		trail	execute
retry		unwind_trail	proceed
trust		tidy_trail	
		unify	

TABLE 3.1. *WAM* Commands and Ancillary Functions Implemented.

3. Individual Commands

The individual commands are documented in [1]. Happily, the commands as documented were coded virtually verbatim. There were two major design innovations required: *(i)* the `call-proceed` *versus* `retry` mechanism and *(ii)* the calling of `C` functions directly.

3.1. Standard WAM Commands

The commands implemented in the project are shown in Table 3.1. These represent the commands defined in [1] except for the indexing commands. Since those commands are primarily for optimization it was decided to postpone implementation. It turns out that these commands are implemented virtually verbatim from [1]. The major changes involve the "microcode" available and some decisions to turn descriptions of common operations into subroutines.

3.2. The Compiled Clauses

A compiled clause is a `C` function. The clause expects two arguments: The *dispatch state* and a *label number*. The dispatch state is define in reference to the standard *box model*. That is, the dispatch state indicates whether this call is an initial call, retry, proceed, *etc.* The label number is assigned by the assembler and is interpreted by a set of simple `switch` statements, one for each dispatch state.

The clauses are named according to the `clause`, `call`, and `execute` statement statements. The entry points themselves are protected by a `clause` data type that carrys debugging information as well as the entry point.

3.3. The Dispatcher Program—Implementation of Control

```
CPA := A                        % Save current A-bank
A := @(nextA)                   % Set global A-bank to call list
CP := nextstatement             % Save return statement number
CF := @(current function)       % Save current entry point
num_of_args := n                % Set number of arguments
E->Nargs := n                   % Save previous number of arguments
Qa1 := A                        % Set up to grab...
Qa2 := CLAUSE(Qa1[1])           % The entry point
trace                           % trace call if tracing on
B0 := B                         % Set up neck cut
Dispatch := initial_call        % Set dispatcher to call
longjmp(dispatch_longjmp,0)     % Do long jump
```

FIGURE 3.1. Call Processing At *longjmp*

The mechanism for `call` or `execute` relies on the `longjmp` mechanism found in *Unix*. Before settling on this mechanism, we had invented a complicated *call-return* mechanism. This disappears in the `longjmp` version.

How should we define the program counter registers `P` and `CP`? In order to make the system work, we define the state to be the pair $(entry - point, jump - table - number)$. In fact, without this complication we might have been able to implement the system as a set of macros only. In pseudocode, the `call` is shown in Figure 3.1. The only processing in the dispatcher is actual call. By not doing the call from the calling procedure we shorten the `C` stack. `proceed` processing is the reverse of `call` processing, just as one would expect. `execute` processing is much simpler than `call` processing since `execute` has a state that need not be saved.

The last standard `WAM` situation to consider is that of *backtracking.* `backtrack` can be called from any of a number of command. The routine described in [1] has been moved to the `dispatcher`, and the call to `backtrack` has been kept for uniformity and error checking. The processing is effectively that of [1].

4. The Assembler

The assembler is a simple `yacc` assembler with very few amenities since it is not meant for human use. The basic idea is to simplify the development of a system. A true compiler could generate code directly, but having the assembler allows for others to use. Some features:

- The assembler expects to read in a `nucleus`. The `nucleus` consists primarily of the `registers` used. This generates the `extern` statements.
- The assembler also expects to use a `prelude` to load information about the other elements in the system. This effectively loads the symbol table.
- Within either the `nucleus` and the `prelude`, one can define
  - keyword `fundamental` to load `C` definitions,
  - routines in `f/n` format,
  - registers,
  - structures, and
  - tags.

```
clause ebmain/0
 permanent  Name, Cont, Nargs,HeadStart,A1,A2,A3,
           A4,A5,number1,number2, constant128
 start
 trace   1
 try_me_else L2
   allocate
   print "Got to ebmain/0\n"
  INTTOCELL 128, constant128
  WRITETAG constant128,CON
  INTTOCELL 2,Nargs
  INTTOCELL 0,Cont
* set_structure localdummy/2
  put_constant 16,A1
  put_constant 32,A2
  call    localdummy/2
  print   "ebmain...return to ebmain/0\n"
  halt
L2:
  trust_me
  print "something failed\n"
  halt
end
```

FIGURE 3.2. An Example Program.

The style of clause input is similar to that used in [1] and a sample program is shown in Figure 3.2. Each clause is to be the processing for the particular arity as defined in the `clause` statement. After the `clause` but before the `start` statement come the declarations. Declarations are local to the clause and include the elements of a `prelude` plus `permanent` and `temporary` names. These are variables that are on the stack (`Y` and `X` bank respectively). Notice at the statement marked (*) a new command `set_structure`. This marks the beginning of the building of a call list and guarantees the initialization of the `A` bank. Other non-standard commands such as `trace`, `print`, and various genre of `dump` are built into the assembler.

One last issue: dealing with `C` routines that are not generated by the compiler. It is clearly necessary to be able to manipulate memory values outside the `WAM` model. We have chosen to make this a separate extension. The concept is as follows: the argument (`A` bank) is computed as with any other call. Once constructed, the arguments are moved to the `W` bank. This is a vector guaranteed to be out of the `WAM`'s memory model. The `W` bank has special operations as outlined in Table 4.1. This extension is sufficient to gain access to `C` and `Fortan` routines.

5. Conclusion

In retrospect, the development of this assembler was quite simple. Although [1] is very clear there are several subtle points that needed to be worked out. The

`openWbank`	Move `Abank` to `Wbank`
`closeWbank`	Move `Wbank` to `Abank`
`naked`	Remove the tag field
`invokefun`	Invoke as function, set `Wresult`
`invokesub`	Invoke as procedure
`protect`	Place a tag onto the `W` element
`move_W_value`	Forcibly assign `W` value to variable
`W_result`	Make the `Wresult` value.
`fail_if_zero`	Test `Wresult`; fail if zero
`fail_if_not_zero`	Test `Wresult`; fail if not zero

TABLE 4.1. Special Commands for *C* External Procedures.

actual development reported here took about one year but three major iterations. Most of this time was spent in testing various strategies to make backtracking fit in the `C` scheme of memory management. Since `WAM` $\neq$ `Prolog`, the assembler has no current provisions of knowledge base operations. These operations are being held in abeyance pending more detailed development of the numerical language.

We are currently rewriting the assembler to remove the last vestiges of the `i`-machine.

A higher-level language is being defined based on the use of *guarded* commands[4]. The guarded `if` and `do` introduced by Dijkstra fall naturally on top of the backtracking paradigm of the `WAM`. We have extended the assembler to support these two statements.

REFERENCES

1. Hassan Aït-Kaci. *Warren's Abstract Machine.* MIT Press. 1991.
2. Errett Bishop and Douglas Bridges. *Constructive Analysis.* Springer Verlag. 1985.
3. K.L. Clark and S.-A. Tarnlund. *Logic Programming.* Academic Press. 1982.
4. Wim H. Hesselink. *Programs, Recursion and Unbounded Choice.* Cambridge Tracts in Theoretical Computer Science *27.* Cambridge University Press. 1992.